ECE 438: Computer Design
Electrical and Computer Engineering
University of New Mexico

Instructor: Andrew Targhetta

Lab 5

A Pipelined MIPS Processor, Part II

Due: May 16th, 2020

Introduction

The objective of the last lab this semester is to complete our pipelined MIPS processor by creating the top-level VHDL component that will instantiate the individual components built in prior labs and implement the remaining portions of the datapath shown in Figure 1.

Recall that we will support the following instructions:

- Arithemtic — ADDU, ADDIU, SUBU
- Logical — AND, ANDI, NOR, OR, ORI, XOR, XORI
- Shift — SLL, SRL, SRA
- Comparison — SLTU, SLTIU
- Loads and stores — LW and SW
- Branches and Jumps — BEQ and J
- Load Upper Immediate — LUI

Background

Figure 1 shows the block diagram of the 5-stage, pipelined MIPS processor you will complete in this lab. As shown, your processor will use forwarding to resolve data hazards and separate memories to resolve structural hazards. Control hazards will be resolved using a branch delay slot in combination with the *assume-Not-Taken* branching scheme discussed in lecture.

In prior labs, we developed the register file, instruction decode unit, ALU decode unit, and ALU for the single-cycle MIPS design. If your *single-cycle* processor correctly executed the MIPS assembly in the provided testbench, then these individual components should be ready to use in your *pipelined* MIPS processor with the exception of modifying the register file to perform writes on the falling edge of the clock. In lecture, we discussed a form of forwarding around the register file in which writes to the register file happen in the first half of the clock cycle, while reads from the register file happen in the second half of the clock cycle. We can easily accomplish this by triggering writes into the register file on the falling edge of the clock. Recall that this particular forwarding resolves a hazard in which the producing instruction writes to the register file on the same cycle as the consuming instruction reads from the register file.

In Lab 4, we developed the pipeline control unit and forwarding unit for handling the control and data hazards introduced with pipelining. Some troubleshooting of those components might be required in this lab as we did not test them in the last lab.

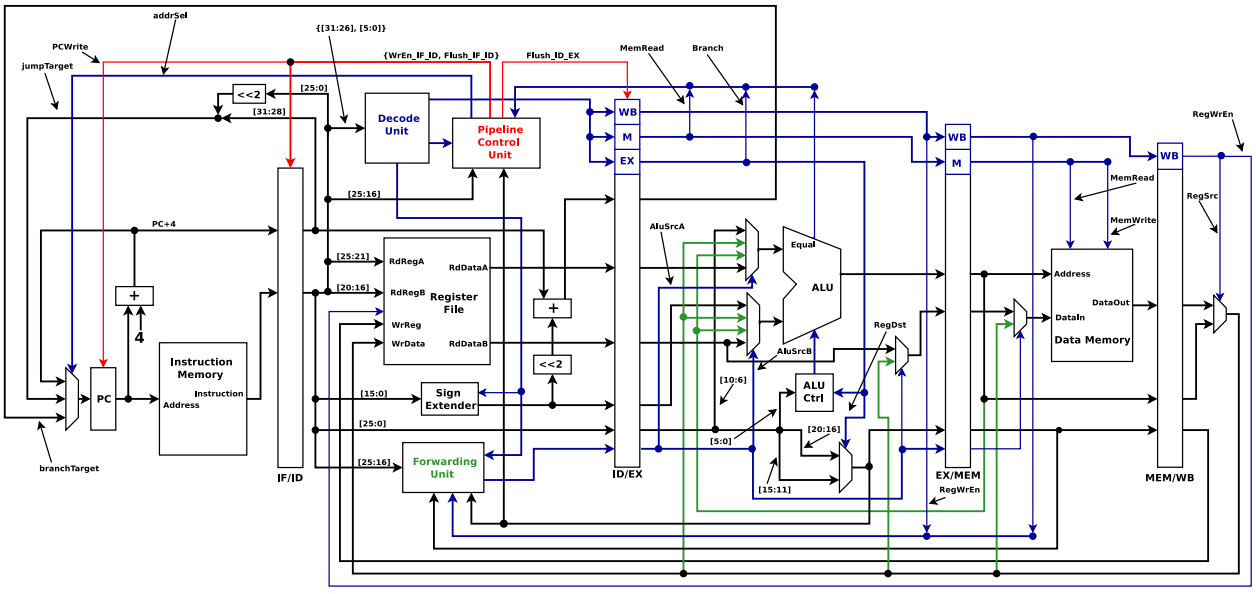

Figure 1: MIPS Block Diagram

Procedure

1. Modify your register file to perform writes on the falling edge of the clock.

2. Describe the pipelined datapath depicted in Figure 1 using the skeleton code provide below as a starting point. Pay close attention to the signal naming convention described in the comments at the top of the source file. This convention will make it easy to read the flow of data through the processor and avoid signal naming issues as many signals exist in more than one pipeline stage.

```
1  ----------------------------------------------------------------------------
   -- This VHDL source file describes the basic 5-stage pipelined processor
3  -- discussed in David Patterson and John Hennessy's Computer Organization and
   -- Design textbook. Instruction processing goes through the following stages:
5  --IF ---  Instruction fetch: Here instructions are fetch, one per clock cycle,
   --        from the memory pointed to by the Program Counter (PC).
7  --ID ---  Instruction decode: The instruction fetched in the prevous stage is
   --        decoded into a number of control signals that flow down the pipe.
9  --        Also, the register file is read in this stage. Data and control
   --        dependencies are detected and handled in this stage, either with
11 --        forwarding or interlocking.
   --EX ---  Execute: The Arithmetic-Logic Unit (ALU) is in this stage. Also,
13 --        branches are resolved in the execute stage.
   --MEM --- Memory: In this stage, memory is accessed. For a load, memory is
15 --        read, while for a store, memory is written.
   --WB ---  Write Back: This is the final stage of the pipeline. Here the
17 --        register file is written.
   --
19 --The logic described in this src file attempts to follow the natural flow of
   --the pipeline. PC logic and other instruction fetch related HW is described
21 --at the beginning, while Write-back logic is at the end.
   --
23 --The signals in the pipeline attempt to adhere to the following format:
   --prefix_signal_name_suffix
25 --A prefix identifies the location within the pipeline where a given signal
   --comes from. If two stages are specified for the prefix, then the signal
27 --comes from the pipeline register between those two stages. For example,
   --ID_EX_load is the load signal from the ID/EX pipeline register. If only one
29 --stage is specified in the prefix, the signal originates from the given
   --stage. For example, ID_load originates from the decode unit within the ID
31 --stage. If no prefix exists, that signal is assumed to be contained within
   --only one pipeline stage.
33 --A suffix indicates the destination of a signal and is added to a signal name
   --only when necessary. Most signals are used in multiple stages and therefore
35 --have no specific destination. However, some signals have the same signal
   --name but are expected to used in different stages. For example,
37 --DataMemForwardCtrl_EX and DataMemForwardCtrl_MEM are different signals,
   --serving similar functions but destine for different stages.
39 ----------------------------------------------------------------------------
```

```
41 library ieee;
   use ieee.std_logic_1164.all;
43 use ieee.std_logic_unsigned.all; --needed to describe
   use ieee.std_logic_arith.all;    --arithmetic
45 ----------------------------------------------------------------------------

47 entity MIPSpipeline is

49 port( Clk, Rst_L: in std_logic;
         PC: out std_logic_vector(31 downto 0);
51       Instruction: in std_logic_vector(31 downto 0);
         DataMemAddr: out std_logic_vector(31 downto 0);
53       DataMemRdEn, DataMemWrEn: out std_logic;
         DataMemRdData: in std_logic_vector(31 downto 0);
55       DataMemWrData: out std_logic_vector(31 downto 0)
   );
57 end MIPSpipeline;

59 ----------------------------------------------------------------------------

61 architecture MIPSpipeline of MIPSpipeline is

63 -- Component declarations go here...

65     component RegisterFile is
       port( RdRegA: in std_logic_vector(4 downto 0);
67           --insert code here for read port B
             WrReg: in std_logic_vector(4 downto 0);
69           Clk: in std_logic;
             --something else is missing here...
71           WrData: in std_logic_vector(31 downto 0);
             RdDataA: out std_logic_vector(31 downto 0);
73           --insert code here for read port B
       );
75     end component;

77 -- Declare signals and variables here...

79 -- Typically, the code is easier to read if signals are
   -- declared in the order they appear below...
81
   -- Instruciton fetch signals
83 signal IF_ID_Instruction: std_logic_vector(31 downto 0);
   -- Pipeline control signals
85 signal Flush_IF_ID, WrEn_IF_ID: std_logic;

87 -- Instruction decode signals
   signal ID_Rs, ID_Rt: std_logic_vector(4 downto 0);
```

```
89
   -- Execute stage signals
91 signal Shamt: std_logic_vector(31 downto 0);
   signal AluInA: std_logic_vector(31 downto 0);
93 signal AluInB: std_logic_vector(31 downto 0);
   signal EX_AluResult: std_logic_vector(31 downto 0);
95 signal EX_Equals: std_logic;
   signal EX_RegDst: std_logic;
97 signal EX_DataMemWrData: std_logic_vector(31 downto 0);
   signal EX_WrReg: std_logic_vector(4 downto 0);
99
   --EX/MEM pipeline registers
101 signal EX_MEM_MemRdEn, EX_MEM_MemWrEn: std_logic;
   signal EX_MEM_RegSrc: std_logic;
103 signal EX_MEM_RegWrEn: std_logic;
   signal EX_MEM_AluResult: std_logic_vector(31 downto 0);
105 signal EX_MEM_DataMemWrData: std_logic_vector(31 downto 0);
   signal EX_MEM_WrReg: std_logic_vector(4 downto 0);
107 signal EX_MEM_DataMemForwardCtrl_MEM: std_logic;

109 --MEM/WB pipeline registers
   signal MEM_WB_RegSrc: std_logic;
111 signal MEM_WB_RegWrEn: std_logic;
   signal MEM_WB_DataMemRdData: std_logic_vector(31 downto 0);
113 signal MEM_WB_AluResult: std_logic_vector(31 downto 0);
   signal MEM_WB_WrReg: std_logic_vector(4 downto 0);
115
   -- Write back stage signals
117 signal WB_RegWrData: std_logic_vector(31 downto 0);

119 ---------------------------------------------------------------------------
   begin
121
   ------------------------------------------------
123 -- Instruction Fetch logic

125 -- insert PC logic here, marking the beginning of the fetch stage

127 -- IF_ID pipeline registers mark the end of the fetch stage
     process(Clk)
129  begin --rising edge triggered logic
       if(Clk'event and Clk='1') then
131      if(Flush_IF_ID='1') then --active high, synchronous reset
           IF_ID_Instruction <= (others => '0');
133        --Add more registers here
         elsif(WrEn_IF_ID='1') then
135        IF_ID_Instruction <= Instruction;
           --Add more registers here
137      end if;
```

```
        end if;
    end process;
-----------------------------------------------------
-- Instruction Decode logic

    --break up instruction into parts for readability
    ID_Rs <= IF_ID_Instruction(25 downto 21);
    ID_Rt <= IF_ID_Instruction(20 downto 16);

    --instantiate the register file
    --instance_name: component_name
    -- use nominal port map as opposed to positional!
    Registers: RegisterFile
        port map(RdRegA => ID_Rs,
                 RdRegB => ID_Rt,
                 WrReg => MEM_WB_WrReg,
                 Clk => Clk,
                 WrData => WB_RegWrData,
                 --okay you get the point...
        );

-----------------------------------------------------
-- Execute stage logic

    --zero extend the shift amount field
    Shamt(4 downto 0) <= ID_EX_Instruction(10 downto 6);
    Shamt(31 downto 5) <= (others => '0');

    -- ALU source mux for OpA
    --The WITH/SELECT/WHEN construct is great for muxes
    with ID_EX_AluSrcA select
        -- the zero extended shamt field
        AluInA <= Shamt when "00",
        -- the bypass path from the write back stage
                  WB_RegWrData when "01",
        -- the bypass path from the memory stage
                  EX_MEM_AluResult when "10",
        -- the register file output
                  ID_EX_RdRegA when others;--avoids latches!

-- ALU should be instantiated somewhere here...

-- EX/MEM pipeline registers here...

-----------------------------------------------------
-- Memory stage logic

-- DataMemWrData mux here...
```

```
187 -- MEM/WB pipeline registers here...

189 end MIPSpipeline;
    ---------------------------------------------------------------------------
```

3. Using the provided testbench, verify the correct operation of your processor.

1 Deliverables

1. Email to `adtargh@unm.edu` or submit via Learn all VHDL source files with comments .

   Note: Code submitted without adequate comments will not be graded!

2. Answer the following questions:

   (a) In our pipeline, can the data hazard below be resolved with forwarding alone? Why or why not?

   ```
   lw $t0, 0($t1)
   sw $t3, 0($t0)
   ```

   (b) Will the pipeline control unit *slip* the pipeline in the case of the instruction sequence above? Why or why not?

   (c) Which components in Figure 1 does the test bench file, `MIPSpipeline_tb.vhd`, provide?

   (d) Determine the CPI of the test code running on your pipelined processor. Compare this CPI to that of the single-cycle processor.